Examen final – LOG3000 – Hiver 2015

- Samedi le 18 avril 2015.
- Durée : 13h30 à 16h00 (total 2h30).
- Local : A-416.2.
- Total des points : 20.
- Pondération de l'examen dans la note finale : 40%.
- Sans documentation, sans calculatrice.
- Contact du chargé de cours : mathieu.lavallee@polymtl.ca, poste 7187.
- Les images peuvent être en tons de gris.

#1. Question sur le travail en équipe et la résistance au changement (3 points)

Vous travaillez pour une entreprise faisant des jeux vidéo. L'entreprise possède une équipe d'une douzaine de développeurs ainsi qu'une équipe de taille variable de testeurs. Les développeurs sont des employés permanents de l'entreprise. Comme il n'y a pas toujours un produit prêt à être testé, les testeurs sont des employés à contrat engagés pour tester un jeu particulier. Les deux équipes travaillent dans le même bâtiment, mais dans des locaux séparés.

Le problème est qu'il y a beaucoup de conflits entre l'équipe de développeurs et l'équipe de testeurs. Voici ce que chaque équipe a à dire sur le sujet :

- Selon les développeurs : Les testeurs sont incompétents. Ils se concentrent sur des détails accessoires et laissent passer des bogues critiques. On doit constamment surveiller leur travail afin d'être sûrs qu'ils travaillent sur les bonnes sections du jeu.
- Selon les testeurs : Les développeurs sont hautains et irrespectueux. Ils ne fournissent pas de documentation sur le code qui a été écrit ou modifié d'une version à l'autre. Il est difficile pour nous de savoir quoi faire.

Sur la base de ces informations :

- Proposer une activité ou une pratique qui permettrait de résoudre le conflit actuel (1 point).

- Justifiez votre choix en indiquant l'impact que pourrait avoir cette activité ou cette pratique (1 point).

- Cela fait dix ans que l'entreprise fonctionne de cette manière. Les développeurs sont convaincus que le problème est du côté des testeurs, alors que les testeurs sont convaincus que le problème est du côté des développeurs. Quelle approche utiliseriez-vous afin de convaincre les deux équipes de changer leur façon de faire (1 point) ?

Solution possible :

Une approche possible serait d'avoir une meilleure gestion du changement qui documente clairement les changements effectuées d'une version (*baseline*) à l'autre. Il faudrait encourager les développeurs à mieux documenter leurs soumissions (*commits*) de code. À chaque version, cette documentation des soumissions de code serait compilée et remise aux testeurs.

L'impact immédiat serait que les testeurs sauraient exactement qu'est-ce qui a été changé depuis la dernière version. Ils sauraient exactement quels tests effectuer. Ils pourraient faire des tests de régression plutôt que de tester l'ensemble du logiciel à chaque fois, et donc se faire reprocher de ne pas travailler sur les bonnes sections du jeu. L'impact sur les tests au complet serait qu'on testerait immédiatement les changements effectués, donc les bogues majeurs seraient trouvés plus rapidement.

Je ferais un projet pilote avec un ou deux développeurs parmi les plus motivés par le changement. Je montrerais comment les changements faits par ces développeurs sont mieux documentés, et donc mieux testés que ceux des autres. J'accumulerais des faits qui démontrent la pertinence du changement, et je m'assurerais que j'ai plus de données qui supportent le changement que d'arguments contre le changement.

#2. Question sur l'analyse quantitative de processus (2 points)

Vous êtes gestionnaire d'un projet de développement logiciel. Le projet est maintenant bien avancé, et beaucoup de code a été écrit. Depuis le début du projet, vous saisissez plusieurs métriques sur le code afin d'en suivre sa qualité.

Récemment, un module semble présenter certains problèmes au niveau de son architecture. Ce module contient cinq classes qui interagissent entre elles. Voici les données que vous avez pour ces cinq classes.

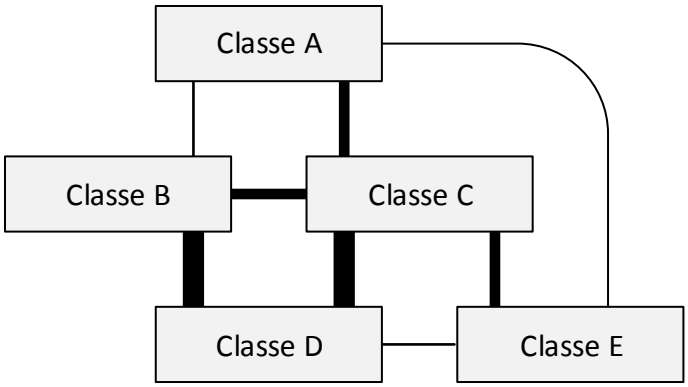


Figure 1. Classes et les appels de fonction entre elles. Plus le lien entre deux classes est épais et plus il y a des appels entre les deux.

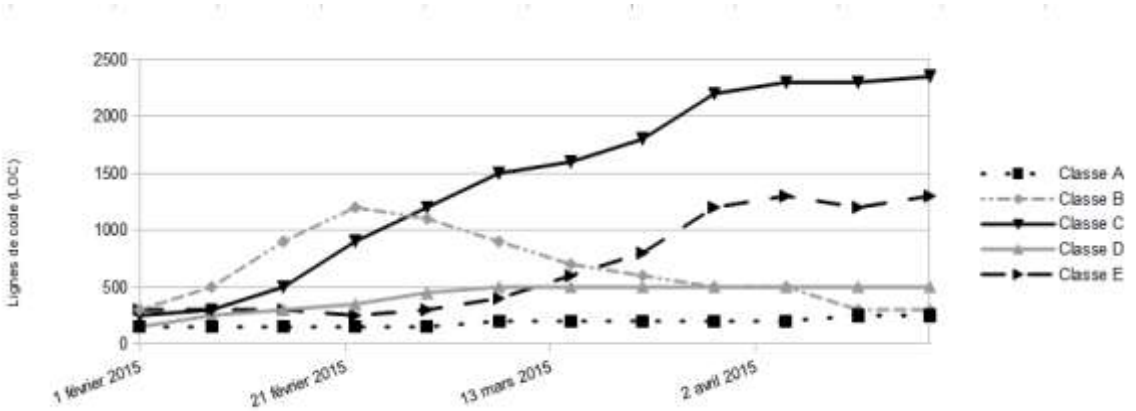


Figure 2. Nombre de lignes de code (LOC) en fonction du temps pour les cinq classes.

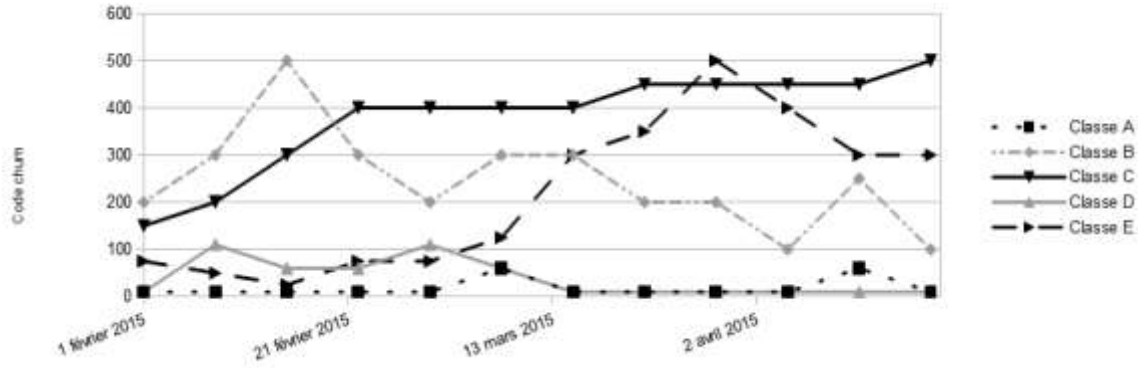


Figure 3. "Code churn" (lignes de code ajoutées, lignes de code enlevées, lignes de code modifiées) en fonction du temps pour chaque classe.

La question que vous vous posez est donc la suivante :

- Est-il temps de faire une refactorisation (*refactoring*) des classes de ce module ? Laquelle ou lesquelles de ces classes devraient être corrigées ? (1 point)
- Justifiez votre choix de classes sur la base des données présentées. (1 point).

Solution possible :

Oui, je ferais une refactorisation, en particulier des classes C et B.

La classe C a gonflé en volume, au point d'avoir près de 2500 lignes de code. C'est beaucoup pour une seule classe. De plus, à chaque semaine on change autour de 400 lignes de code de cette classe, ce qui laisse supposer que tous les problèmes sont liés à cette classe.

Le profil de la classe B semble indiquer que son code a été transféré à la classe C, car elle a diminué de taille alors que l'autre augmentait. Il y a peut-être eu des pressions calendaires qui ont forcé les développeurs de déplacer du code d'une classe à une autre afin de travailler plus vite. Il est probable que ce changement a nui à la modularité du code.

#3. Question sur la discipline d'implémentation (3 points)

Vous venez de terminer votre projet intégrateur de 3^{e} année. Vous discutez avec des collègues dans une autre équipe, et ceux-ci semblent particulièrement découragés. Ils ont réussi à finir le projet, mais il a fallu faire plusieurs nuits blanches en fin de projet afin de finaliser l'implémentation. En conséquent, très peu de tests ont été faits, des bogues majeurs sont apparus durant l'évaluation, et vos collègues n'ont pas eu une très bonne note.

Vos collègues vous présentent le processus utilisé pour leur implémentation. Ils ne comprennent pas pourquoi l'implémentation a été aussi difficile, ni pourquoi autant de bogues se sont retrouvés dans le produit final. Ils ont utilisé une approche TDD (*test driven development*) qui aurait dû éliminer une grande quantité de bogues, vu que presque tout le code était couvert par des tests unitaires.

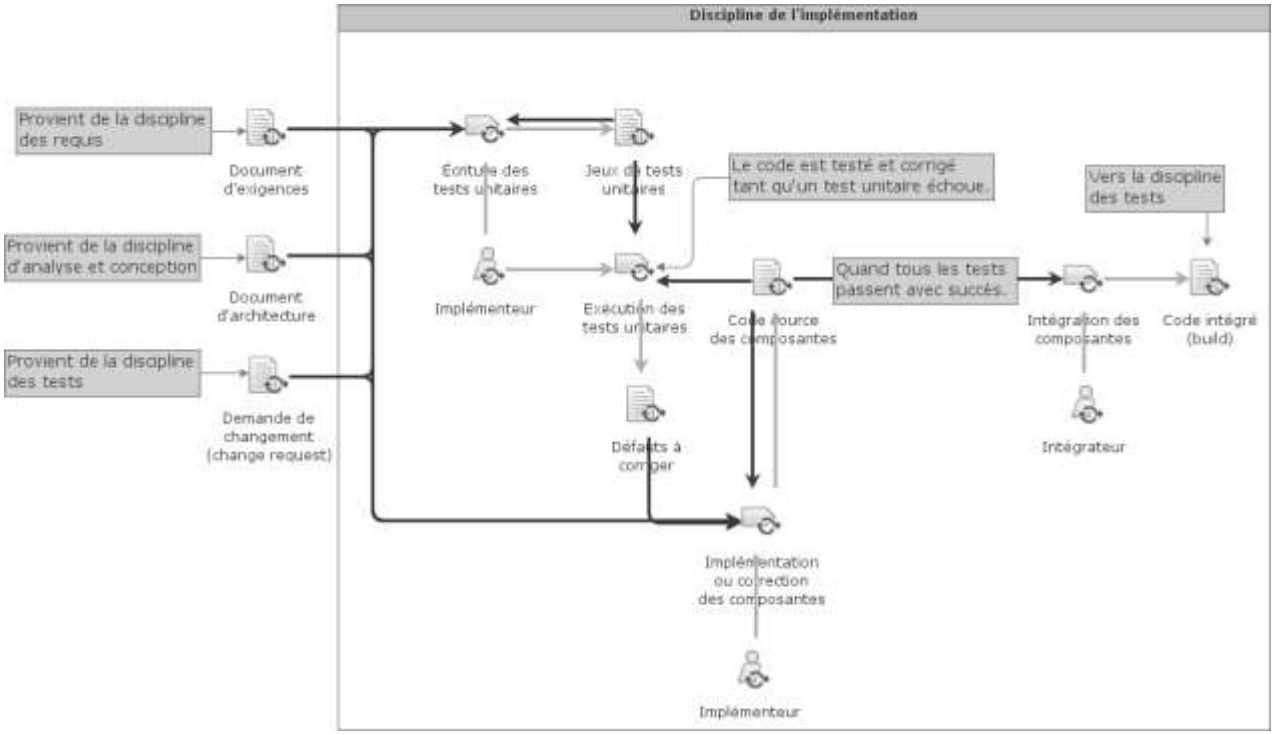


Figure 4 - Processus utilisé pour l'implémentation.

- Sur la base du processus utilisé par vos collègues pour l'implémentation, quel problème pouvez-vous identifier ? (1 point)
- Justifiez votre choix de problème en détaillant l'impact que celui-ci pourrait avoir eu sur le développement logiciel. (1 point)
- Proposez une solution à ce problème. Justifiez votre solution en indiquant l'impact que votre solution aurait sur le problème identifié. (1 point).

Solution possible :

Les tests unitaires, comme leur nom l'indique, se limitent à tester un objet pris individuellement. Un objet qui passe les tests unitaires ne veut pas dire que celui-ci pourra communiquer adéquatement avec les autres objets. Il aurait donc fallu faire des intégrations plus tôt afin de détecter ces problèmes de communication entre objets.

En ne faisant qu'une intégration à la fin, on prend le risque que les objets ne communiquent pas correctement entre eux. Cela peut cause des retards, tels que ceux vécus par l'équipe de nos collègues. Cela peut aussi causer des bogues majeurs qui peuvent être difficile à tester (*race conditions*, etc.) tels que ceux observés par nos collègues.

Je découperais mes exigences en blocs, en utilisant un processus de style Scrum. De cette manière, on s'assure de faire des intégrations fréquentes, et donc d'avoir en main un produit testable. Dans le pire cas, certaines exigences optionnelles ne seront pas complétées, mais au

moins l'essentiel du produit sera construit, intégré et suffisamment testé.

#4. Question sur l'utilisation des standards (3 points)

Une petite entreprise en démarrage (*startup*) vient de subir une expansion rapide. Vous êtes donc engagés par cette entreprise afin de gérer la toute nouvelle équipe d'assurance-qualité. Cette nouvelle équipe reste modeste par contre et inclus seulement vous et deux testeurs.

Comme vous devez monter un processus d'assurance-qualité à partir de zéro, vous observez ce que la norme ISO 12207 recommande. La norme recommande les quatre artéfacts suivants :

- Plan détaillant la stratégie utilisée pour assurer la qualité. Ce plan décrit les méthodes et outils utilisés dans les activités d'assurance-qualité. Le plan décrit aussi les ressources et les échéanciers prévus pour les activités d'assurance-qualité.

- Rapport démontrant que les activités d'assurance-qualité sont exécutées et mises-à-jour. Il s'agit d'un rapport qui fait le suivi de l'avancement des tâches d'assurance-qualité qui se retrouvent dans le plan.

- Rapport indiquant les problèmes ou les non-conformités avec les exigences. Les exigences comprennent les exigences fonctionnelles, les exigences non-fonctionnelles, et les contraintes imposées par le contrat signé avec le client (coût du développement, date limite de livraison, etc.).

- Rapport indiquant que le processus de développement est respecté, que toutes les activités prévues pour le développement sont faites, et que les normes imposées sont respectées. Il s'agit de vérifier que le développement se fait selon les règles de l'art et que l'équipe de développement a les compétences requises.

Comme vos ressources sont limitées, vous ne pourrez pas gérer tous ses artéfacts. Pour chacun des quatre artéfacts présentés, décidez si vous voudriez qu'il soit dans votre processus d'assurance-qualité ou non. Justifiez vos choix de garder ou de rejeter chaque artéfact.

Solution possible :

Le plan d'assurance-qualité sera fort probablement demandé par l'entreprise. Les gestionnaires de l'entreprise voudront savoir à quoi les fonds investis en assurance-qualité vont servir, surtout s'il s'agit d'un nouveau processus.

Si le plan d'assurance-qualité est fait sur un outil comme Microsoft Project, il est possible de documenter le suivi des tâches assez facilement. Je ne ferais pas un rapport exhaustif, mais je m'assurerais que le diagramme de Gantt sur Microsoft Project est toujours à jour. Encore une fois, cela risque d'être demandé par les gestionnaires de l'entreprise, alors nous n'aurons probablement pas le choix.

Pareillement pour le rapport sur les problèmes. Je ne ferais pas un rapport exhaustif sur le sujet, mais j'utiliserais plutôt les outils actuels utilisés. Il est probable que l'équipe utilise déjà des outils comme Bugzilla pour suivre les bogues. Il serait possible de documenter les problèmes fonctionnels et non-fonctionnels directement sur un outil de ce genre. Les problèmes de gestion pourraient être adressés lors des réunions de suivi des développeurs. Il est important que les problèmes soient documentés afin de s'assurer qu'ils sont corrigés, mais surtout il est important que les problèmes trouvés parviennent aux développeurs. C'est pourquoi il est préférable d'utiliser un outil familier pour les développeurs plutôt qu'un rapport qui ne sera probablement pas lu.

Je crois que le dernier rapport est important. Par contre, le rapport peut être fait assez rapidement. Par exemple, il y a fort probablement un diagramme de Gantt des tâches effectuées par les développeurs. On peut vérifier que le diagramme de Gantt couvre toutes les activités prévues. Cet artéfact permet de mesurer la différence entre le travail planifié et le travail réel. Il est important, surtout lorsque les dates limites approchent, de s'assurer que les bonnes pratiques ne sont pas abandonnées.

#5. Question sur la discipline de gestion de configuration et du changement (2 points)

Le développement à code ouvert (*open source*) utilise fréquemment une pratique particulière, la revue de code, dans leurs processus de développement logiciel. Cela fonctionne généralement de la manière suivante :

1. Un premier développeur volontaire soumet une modification de code grâce à un outil comme *Git*.
2. Un deuxième développeur proche du projet (parfois appelé *code marshal*, littéralement, policier du code) lit le code soumit grâce à un outil comme *Gerrit*. Le deuxième développeur s'assure que le code soumit est adéquat.
3. Le *code marshal* approuve le code ou bien le retourne au premier développeur pour correction.

Cette pratique peut apporter certains avantages pour tous les projets logiciels, mais il est possible de l'appliquer incorrectement.

- Présentez un avantage pour un projet de développement logiciel d'utiliser cette approche. Vous devez présenter un avantage immédiat de la pratique (qu'est-ce qui est amélioré par la pratique) et aussi un avantage global (en quoi la pratique améliore le développement logiciel au complet).
- Qu'est-ce qui pourrait faire que la pratique soit mal appliquée ? Si vous deviez gérer ce genre de pratique, qu'est-ce que vous surveilleriez afin de vous assurer que la pratique est bien faite ?

Solution possible :

Un avantage immédiat est qu'on s'assure que le code est vérifié pour sa conformité au guide de programmation et qu'il n'y a pas de problèmes architecturaux majeurs. Cela assure qu'il y a au moins qui lisent chaque ligne de code, ce qui peut augmenter la compréhensibilité et donc la maintenabilité du code. Un code plus maintenable peut permettre de sauver du temps, parce qu'on perd moins de temps à lire et à comprendre le code lorsqu'on veut le changer.

Ce genre de pratique peut mener à du *rubber stamping*. C'est-à-dire que la personne qui évalue le code se limite à l'approuver sans le lire. C'est particulièrement un problème en fin de projet lorsque des pressions de finir dans les temps et dans les budgets surviennent. Un moyen de détecter ce problème est de voir s'il y a du code qui ne respecte pas le guide de programmation imposé. Cela peut être fait rapidement grâce à des outils comme FxCop.

#6. Question sur l'apprentissage par problèmes (2 points)

En 1978, le toit du *Hartford Civic Center* s'est effondré, quelques heures à peine après un match de basketball. Il n'y a, heureusement, eu aucune victime. En 1979, le toit du *Kemper Arena* s'est aussi effondré, encore une fois heureusement, sans faire de victimes. Les circonstances des deux effondrements étaient très similaires : Les deux toits utilisaient une structure de type hangar, les deux toits se sont effondrés six ans après leur construction, les deux toits se sont effondrés après des précipitations abondantes, et les deux toits avaient été conçus avec l'aide d'un programme informatique complexe.

Les deux enquêtes sur ces effondrements sont arrivées à des conclusions similaires. Les enquêteurs ont pointé du doigt deux problèmes majeurs dans l'utilisation des logiciels :

1. Les programmes informatiques utilisés ne contenaient pas de bogues, mais ils posaient certaines hypothèses sur la répartition des forces. Les ingénieurs en structure n'étaient pas conscients de ces hypothèses et pensaient que les forces étaient calculées autrement. Cela a causé des erreurs de calculs de l'ordre de 20% pour certaines poutres.

2. Durant la construction des bâtiments, les bâtisseurs ont fait remarquer aux ingénieurs en structure que les toits présentaient une distorsion importante plus grande que prévue. Les ingénieurs ont ignoré les avertissements donnés. Les ingénieurs en structure ont répondu que les ordinateurs ne voyaient pas de problèmes et que les ordinateurs ne pouvaient pas faire d'erreur de calcul.

Pour chacun des deux problèmes donnés :

- Selon vous, quelle est la cause la plus probable de l'apparition du problème ? (1 point)
- Comment feriez-vous pour éviter que ces problèmes se répètent dans l'avenir ? (1 point)

Solution possible :

Pour le problème #1, la cause la plus probable est un manque de formation de l'ingénieur en structure. L'ingénieur devrait être mieux informé des limites de l'outil logiciel.

La solution du problème #1 serait de rendre plus évidentes dans le logiciel les hypothèses utilisées. La manière dont la répartition des forces est calculée devrait être clairement indiquée à l'ingénieur en structure.

Pour le problème #2, la cause est probablement due à un manque d'humilité de l'ingénieur en structure. Un simple examen visuel de la distorsion physique aurait permis de voir que celle-ci étaient loin de ce que l'ordinateur avait calculé.

La solution du problème #2 serait de mettre son orgueil de côté et de refaire manuellement les calculs. Étant donné que la structure présentait une distorsion importante, il aurait été bénéfique de simplement vérifier que la distorsion observée était similaire à la distorsion calculée.

#7. Question sur la modélisation de processus (5 points)

Vous travaillez pour une usine qui opère un procédé chimique complexe. Le procédé nécessite des produits chimiques volatiles et explosifs, alors l'essentiel des manipulations sont faites par des machines gérées par un logiciel. L'automatisation du procédé est justifiée par un objectif de diminuer le risque d'une erreur humaine.

Il y a quelques jours, un bogue dans le logiciel introduit après une mise-à-jour du code a causé un déversement accidentel de produits chimiques. Cet accident a fait réaliser aux gestionnaires de l'usine l'impact que le logiciel peut avoir sur le procédé chimique. Les gestionnaires ont donc décidé de formaliser les pratiques de mise-à-jour de code, et vous a donc commandé de monter un processus gérant les demandes de changement au logiciel.

Dans le contexte de l'usine, les demandes de changements (*change request, CR*) sont peu fréquentes. Il peut se passer plusieurs mois avant qu'un changement soit poussé en production (i.e. sur les machines gérant le procédé chimique). Les demandes de changement sont de trois types :

- Bogues à corriger : Les opérateurs qui supervisent le procédé découvrent parfois des problèmes. La majorité des problèmes ont des causes matérielles (ex. : capteur défectueux), mais il arrive parfois que la cause provienne du logiciel.

- Changements dans les machines : Le procédé change de temps en temps. De nouvelles machines sont introduites ou de veilles machines sont enlevées afin de supporter le nouveau procédé. De plus, il arrive parfois qu'une machine tombe en panne et qu'elle doive être remplacée par machine de marque différente.

- Mise-à-jour des librairies : Le logiciel utilise plusieurs librairies développées par d'autres entreprises. Ces librairies sont régulièrement mises-à-jour. La majorité de ces

mises-à-jour sont cependant ignorées parce qu'elles touchent des parties des librairies qui ne sont pas utilisées par l'usine.

L'entreprise possède aussi un simulateur de l'ensemble du procédé chimique. Ce simulateur permet de détecter des bogues majeurs avant que ceux-ci soient appliqués sur le procédé chimique réel.

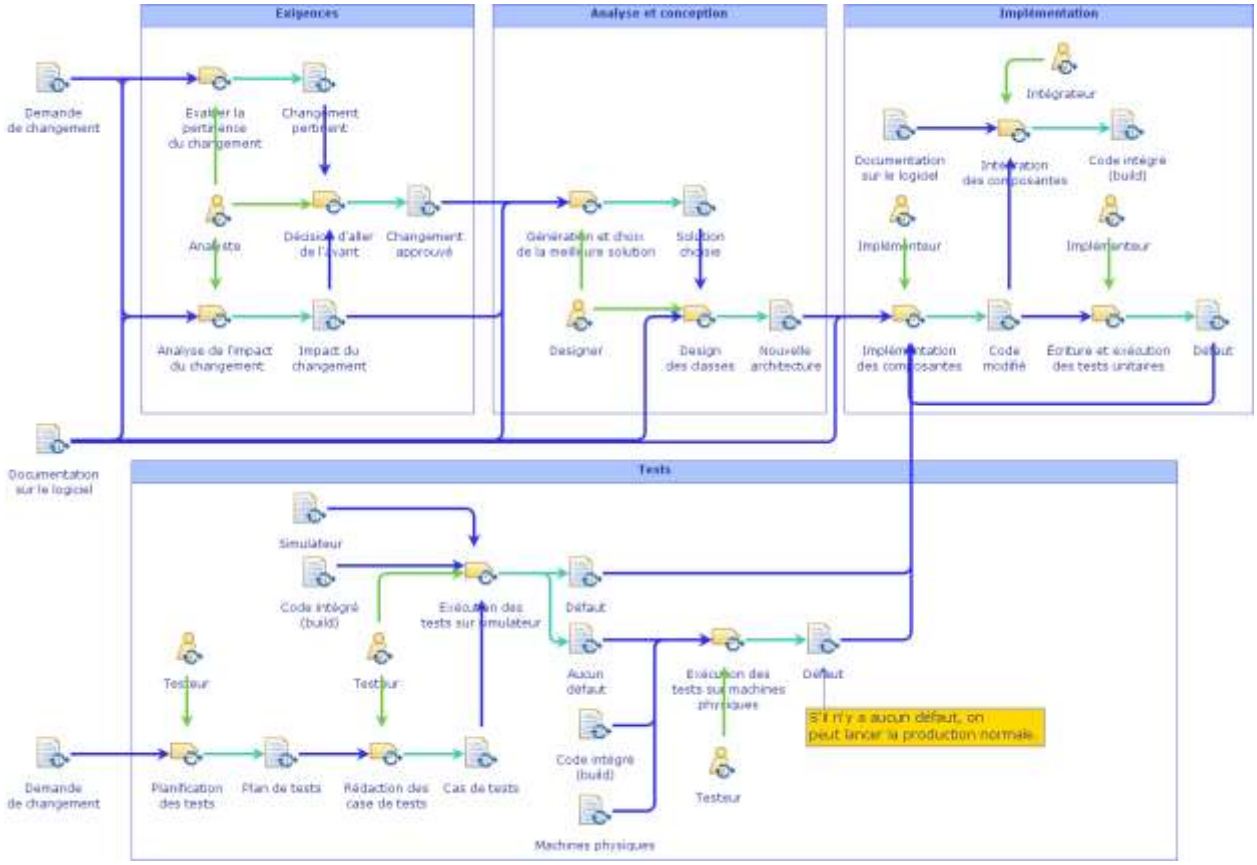
Vous devez donc monter un processus de maintenance selon le langage SPEM 2.0 (le langage de ProcessEdit) qui répond aux objectifs suivants :

- Le processus doit prendre en entrée une demande de changement (*change request*) et doit avoir en sortie le code modifié implanté dans les machines physiques.
- Le processus doit se limiter aux activités de modifications du logiciel. Les problèmes matériels sont gérés par une autre équipe.
- Le processus doit minimiser le plus possible le risque qu'un bogue se trouve dans les machines physiques. Pour l'usine, ce risque est critique : Une panne pourrait causer des délais de production coûteux, un accident pourrait endommager des machines très dispendieuses, et une catastrophe pourrait causer des pertes de vies humaines.

Solution : Il y a plusieurs solutions possibles. Pour avoir tous les points, la solution doit :

- Le processus doit faire une vérification de la pertinence de la demande de changement au niveau des requis (1 point).
- Le processus doit avoir des activités de design. Une activité d'analyse d'impact peut être acceptable (0.5 point).
- Le processus doit avoir toutes les activités d'implémentation, soit de l'écriture de code (0.5 point), l'utilisation de tests unitaires (1 point), et de l'intégration (0.5 point).
- Le processus doit avoir deux types de tests (0.5 point). Un de ces types de test doit être un test sur simulateur (1 point).

Voici un exemple de processus répondant à ces critères :


Exigences
Analyse et conception
Implémentation
Demande de changement
Évaluer la pertinence du changement
Changement pertinent
Analyste
Décision d'aller de l'avant
Changement approuvé
Analyse de l'impact du changement
Impact du changement
Génération et choix de la meilleure solution
Solution choisie
Designer
Design des classes
Nouvelle architecture
Intégrateur
Documentation sur le logiciel
Intégration des composantes
Code intégré (build)
Implémentateur
Implémentateur
Implémentation des composantes
Code modifié
Écriture et exécution des tests unitaires
Défaut
Documentation sur le logiciel
Tests
Simulateur
Code intégré (build)
Exécution des tests sur simulateur
Défaut
Aucun défaut
Exécution des tests sur machines physiques
Défaut
S'il n'y a aucun défaut, on peut lancer la production normale
Testeur
Testeur
Testeur
Demande de changement
Planification des tests
Plan de tests
Rédaction des case de tests
Cas de tests
Code intégré (build)
Machines physiques